WOLFRAM LANGUAGE

Things to Try

This document is a live Wolfram Notebook that mixes text and code.
Run any piece of code by clicking inside the code, then press shift enter .

Compute 2+2 (just click in the input below and press shift enter):

↓

```
In[63]:= 2 + 2
```

Make a list of numbers up to 100:

```
In[64]:= Range[100]
```

Make a list of the first 100 prime numbers:

```
In[65]:= Table[Prime[n], {n, 100}]
```

Also plot the values:

```
In[66]:= ListPlot[Table[Prime[n], {n, 100}]]
```

Get a list of common words in English:

```
In[67]:= WordList[]
```

Compute the length of each word:

```
In[68]:= StringLength[WordList[]]
```

Make a histogram of the lengths:

```
In[69]:= Histogram[StringLength[WordList[]]]
```

Take the first letter of each word:

```
In[70]:= StringTake[WordList[], 1]
```

Make a word cloud from the list of first letters:

```
In[71]:= WordCloud[StringTake[WordList[], 1]]
```

Get a list of countries in Europe (type ctrl = to enter natural language):

```
In[72]:= countries in europe
```

Get images of the flags of these countries:

```
In[73]:= EntityValue[countries in europe, "FlagImage"]
```

Plot these flags in a machine-learned “feature space” with “similar” flags nearby:

```
In[74]:= FeatureSpacePlot[EntityValue[countries in europe, "FlagImage"]]
```

Get a list of the capital cities in South America:

```
In[75]:= cities = capital cities in south america
```

Plot them on a map:

```
In[76]:= GeoListPlot[cities]
```

Find an ordering of these cities that gives the shortest tour that visits all of them:

```
In[77]:= tour = FindShortestTour[GeoPosition[cities]]
```

Plot the cities in the order of the shortest tour, joining them up:

```
In[78]:= GeoListPlot[cities⟦Last[tour]⟧, Joined → True]
```

Find the 10 mountains nearest to where the internet thinks you are:

```
In[79]:= mountains = GeoNearest["Mountain", Here, 10]
```

Make a 3D plot of the terrain in a two-mile region around the first of these:

```
In[80]:= ListPlot3D[
          GeoElevationData[GeoDisk[First[mountains], 2 mi]], ImageSize → Full]
```

Get the current image from your computer’s camera, and call it image:

```
In[81]:= image = CurrentImage[]
```

If you do not have a camera on your computer, use this instead:

```
In[82]:= image = ;
```

Detect edges in the image:

```
In[83]:= EdgeDetect[image]
```

Create an interface to interactively reduce the number of colors in an image and also replace the red color with another color:

```
In[84]:= Manipulate[ImageRecolor[ColorQuantize[image , q], Red → c],
          {{q, 32}, 4, 64, 1}, {c, Green}]
```

Load some data from a CSV file, and display it as a dataset:

```
In[85]:= data = Dataset[Import["https://data.nasa.gov/api/views/gh4g-9sfh/rows.csv"]]
```

Make a histogram of the log of all entries in column 5:

```
In[86]:= Histogram[Log[Normal[data[All, 5]]]]
```

Make a word cloud of column 4:

```
In[87]:= WordCloud[data[All, 4]]
```

Here is the same data, but cleaned and structured in the Wolfram Data Repository:

```
In[88]:= ResourceData["Meteorite Landings"]
```

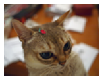
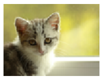
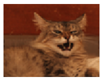
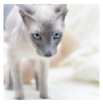
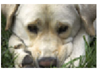
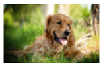
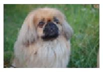
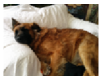
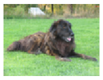
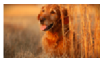
Now things like geo coordinates can immediately be used, here to show where a sample of meteorites landed:

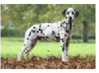
```
In[89]:= GeoListPlot[
          RandomSample[ResourceData["Meteorite Landings"][All, "Coordinates"], 200]]
```

Use machine learning to identify what images are of:

In[90]:= ImageIdentify[{ , , , }]

Classify images by automatically learning from examples:

In[46]:= c = Classify[<|"cat" → { , , , , , , , , , }, "dog" → {, , , , , , , , , }|>]

In[47]:= c[{, , , }]

Find and interpret textual mentions of countries in the Wikipedia article about elephants:

```
In[93]:= countries =
           TextCases[WikipediaData["elephants"], "Country" → "Interpretation"]
```

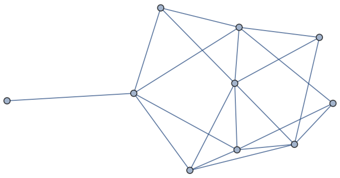
Make a bubble chart illustrating the number of mentions by country:

```
In[94]:= GeoBubbleChart[Counts[countries]]
```

The Wolfram Language is symbolic, so x is just a symbol:

```
In[95]:= x
```

All these things are represented symbolically in the Wolfram Language:

In[96]:= { , Vincent van Gogh PERSON ⋯ ✓, }

Here is a symbolic function called f applied to x:

```
In[97]:= f[x]
```

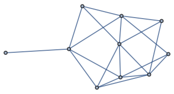
The function Framed displays with a frame around whatever it is applied to:

```
In[98]:= Framed[x]
```

This frames the list of objects:

In[99]:= `Framed[{`

`, Vincent van Gogh PERSON ⋯ ✓, `

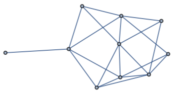


`}]`

This instead puts a frame around each object:

In[100]:= `Map[Framed, {`

`, Vincent van Gogh PERSON ⋯ ✓, `



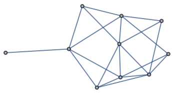
`}]`

The “pure function” puts a random background color behind each object:

In[101]:= `Map[Framed[#, Background → RandomColor[]] &,`

`{`

`, Vincent van Gogh PERSON ⋯ ✓, `



`}]`

Successively apply a symbolic function f to x:

```
In[102]:= NestList[f, x, 10]
```

If the function is Framed, this makes nested frames:

```
In[103]:= NestList[Framed, x, 10]
```

Use a pure function to put a random background color inside each frame:

```
In[104]:= NestList[Framed[#, Background → RandomColor[]] &, x, 10]
```

Create a graph of countries that share a border with Switzerland and their neighbors by nesting the "BorderingCountries" property twice:

```
In[105]:= NestGraph[#["BorderingCountries"] &,
            Switzerland COUNTRY , 2, VertexLabels → Automatic]
```

StringReverse reverses the characters in a string:

```
In[106]:= StringReverse["wolfram"]
```

The reversed string is not the same as the string itself:

```
In[107]:= StringReverse["wolfram"] == "wolfram"
```

This selects words in English that read the same backward and forward:

```
In[108]:= Select[WordList[], StringReverse[#] == # &]
```

The same result for Russian:

```
In[109]:= Select[WordList[Language → "Russian"], StringReverse[#] == # &]
```

You can define rules to give values to symbolic expressions:

```
In[110]:= fac[1] := 1
```

Now fac[1], i.e. factorial of 1, is 1:

```
In[111]:= fac[1]
```

The system does not yet know what fac[10], i.e. factorial of 10, is:

```
In[112]:= fac[10]
```

Define a generic rule to compute the factorial of any integer (represented by the pattern n_Integer):

```
In[113]:= fac[n_Integer] := n * fac[n - 1]
```

Now the rules allow you to compute fac[10]:

```
In[114]:= fac[10]
```

Or fac[1000]:

```
In[115]:= fac[1000]
```

You can also define a rule to catch erroneous inputs:

```
In[116]:= fac[anything_] := "Input is not an integer"
```

```
In[117]:= fac["1.1"]
```

You can define rules for functions of arbitrary structures, here for a pair of elements:

```
In[118]:= myfunc[{x_, y_}] := {1 - 2 x + y, y - x / 4}
```

```
In[119]:= myfunc[{6, 7}]
```

It does not take too much Wolfram code to do pretty sophisticated things:

```
In[120]:= ImageTransformation[jaguar SPECIES SPECIFICATION ••• ✓["Image"], myfunc]
```

Make a simple piece of interactive graphics:

```
In[121]:= Manipulate[Graphics[{Yellow, Disk[], Black, Disk[{0, 0}, r]}], {r, 0, 1}]
```

Publish it to the cloud:

```
In[122]:= CloudPublish[
           Manipulate[Graphics[{Yellow, Disk[], Black, Disk[{0, 0}, r]}], {r, 0, 1}]]
```

Click the URL to visit the interactive website you have created.

This publishes a form interface to the cloud, putting more cats on the internet:

```
In[123]:= CloudPublish[FormPage[{{"breed", "Enter a cat breed:"} → "CatBreed",
             "color" → "Color", "angle" → "Number" → 0}, ExportForm[
              Rotate[Blend[{#breed["Image"], #color}], #angle Degree], "PNG"] &]]
```

You can turn this into an API as well, and get the code to embed it in an external program:

```
In[124]:= EmbedCode[APIFunction[
            {"breed" → "CatBreed", "color" → "Color", "angle" → "Number" → 0},
            ExportForm[Rotate[Blend[{#breed["Image"], #color}], #angle Degree],
              "PNG"] &], "Java"]
```

Where to go next:

- Fast Introduction for Programmers
- *An Elementary Introduction to the Wolfram Language*
- Wolfram U

See some larger programs:

- Code Gallery
- Wolfram Demonstrations Project
- Notebook Archive